[**从内存中加载并启动一个exe**](http://blog.csdn.net/fjclc2008/article/details/19576049)

[windows](http://www.yesmybi.cn/)似乎只提供了一种启动进程的方法：即必须从一个可执行文件中加载并启动。而下面这段代码就是提供一种可以直接从内存中启动一个exe的变通办法。用途嘛, 也许可以用来保护你的exe，你可以对要保护的 exe 进行任意切分、加密、存储，只要运行时能将exe的内容正确拼接到一块内存中，就可以直接从内存中启动，而不必不安全地去生成一个临时文件再从临时文件启动进程。另外这段代码也提供了一种自己写exe外壳的简单途径，如果能配合其它各种外壳技术就更好地保护你的exe文件。原理很简单：就是“借尸还魂”，启动一个僵尸进程（NT下可以是自身程序启动的另一个进程），然后在它运行前将其整个替换成内存中的exe内容，待正式运行后执行的就是你的目标代码了。不过代码中还有一些不尽人意的地方，比如在98下运行会留一个僵尸程序的壳在硬盘上（其实那个僵尸程序本身就是一个完整的可执行程序，直接运行的话只显示一条错误信息然后就退出了）。

另外由于客观条件限制，代码没有经过充分测试，只在XP下进行了一些初步测试：普通exe都能正常运行，upx压缩过的exe绝大多数情况下都能运行，只有在不能卸载僵尸外壳时才有问题（upx压缩过的exe没有重定向表，无法加载到其它地址运行）。如果有bug望告之，如果有更好的方法特别是能解决98下的遗留尾巴的话希望不吝赐教。

原理:

1. 把你的程序读要内存

2. 以 CREATE\_SUSPENDED模式CreateProcess打开svchost.exe

3. 修改svchost.exe页面的属性，然后把要运行的那个程序的内容拷贝到svchost.exe页面

4. 然后再运行 实质想当于是 披着/svchost.exe进程的相关信息/这张皮，而皮里面的肉都被改了

原文来自哪里忘记了，。呵呵

#include <stdio.h>  
#include <windows.h>  
#include <tlhelp32.h>  
#include <stdlib.h>  
//#include "ntpsapi.h"

struct PEHeader  
{  
   unsigned long signature;  
   unsigned short machine;  
   unsigned short numSections;  
   unsigned long timeDateStamp;  
   unsigned long pointerToSymbolTable;  
   unsigned long numOfSymbols;  
   unsigned short sizeOfOptionHeader;  
   unsigned short characteristics;  
};  
typedef struct PEHeader PE\_Header;

struct PEExtHeader  
{  
   unsigned short magic;  
   unsigned char majorLinkerVersion;  
   unsigned char minorLinkerVersion;  
   unsigned long sizeOfCode;  
   unsigned long sizeOfInitializedData;  
   unsigned long sizeOfUninitializedData;  
   unsigned long addressOfEntryPoint;  
   unsigned long baseOfCode;  
   unsigned long baseOfData;  
   unsigned long imageBase;  
   unsigned long sectionAlignment;  
   unsigned long fileAlignment;  
   unsigned short majorOSVersion;  
   unsigned short minorOSVersion;  
   unsigned short majorImageVersion;  
   unsigned short minorImageVersion;  
   unsigned short majorSubsystemVersion;  
   unsigned short minorSubsystemVersion;  
   unsigned long reserved1;  
   unsigned long sizeOfImage;  
   unsigned long sizeOfHeaders;  
   unsigned long checksum;  
   unsigned short subsystem;  
   unsigned short DLLCharacteristics;  
   unsigned long sizeOfStackReserve;  
   unsigned long sizeOfStackCommit;  
   unsigned long sizeOfHeapReserve;  
   unsigned long sizeOfHeapCommit;  
   unsigned long loaderFlags;  
   unsigned long numberOfRVAAndSizes;  
   unsigned long exportTableAddress;  
   unsigned long exportTableSize;  
   unsigned long importTableAddress;  
   unsigned long importTableSize;  
   unsigned long resourceTableAddress;  
   unsigned long resourceTableSize;  
   unsigned long exceptionTableAddress;  
   unsigned long exceptionTableSize;  
   unsigned long certFilePointer;  
   unsigned long certTableSize;  
   unsigned long relocationTableAddress;  
   unsigned long relocationTableSize;  
   unsigned long debugDataAddress;  
   unsigned long debugDataSize;  
   unsigned long archDataAddress;  
   unsigned long archDataSize;  
   unsigned long globalPtrAddress;  
   unsigned long globalPtrSize;  
   unsigned long TLSTableAddress;  
   unsigned long TLSTableSize;  
   unsigned long loadConfigTableAddress;  
   unsigned long loadConfigTableSize;  
   unsigned long boundImportTableAddress;  
   unsigned long boundImportTableSize;  
   unsigned long importAddressTableAddress;  
   unsigned long importAddressTableSize;  
   unsigned long delayImportDescAddress;  
   unsigned long delayImportDescSize;  
   unsigned long COMHeaderAddress;  
   unsigned long COMHeaderSize;  
   unsigned long reserved2;  
   unsigned long reserved3;  
};  
typedef struct PEExtHeader PE\_ExtHeader;

struct Section\_Header  
{  
   unsigned char sectionName[8];  
   unsigned long virtualSize;  
   unsigned long virtualAddress;  
   unsigned long sizeOfRawData;  
   unsigned long pointerToRawData;  
   unsigned long pointerToRelocations;  
   unsigned long pointerToLineNumbers;  
   unsigned short numberOfRelocations;  
   unsigned short numberOfLineNumbers;  
   unsigned long characteristics;  
};  
typedef struct Section\_Header SectionHeader;

struct MZ\_Header  
{  
   unsigned short signature;  
   unsigned short partPag;  
   unsigned short pageCnt;  
   unsigned short reloCnt;  
   unsigned short hdrSize;  
   unsigned short minMem;  
   unsigned short maxMem;  
   unsigned short reloSS;  
   unsigned short exeSP;  
   unsigned short chksum;  
   unsigned short exeIP;  
   unsigned short reloCS;  
   unsigned short tablOff;  
   unsigned short overlay;  
   unsigned char reserved[32];  
   unsigned long offsetToPE;  
};  
typedef struct MZ\_Header MZHeader;

struct Import\_DirEntry  
{  
   DWORD importLookupTable;  
   DWORD timeDateStamp;  
   DWORD fowarderChain;  
   DWORD nameRVA;  
   DWORD importAddressTable;  
};  
typedef struct Import\_DirEntry ImportDirEntry;

struct Fixup\_Block  
{  
   unsigned long pageRVA;  
   unsigned long blockSize;  
};  
typedef struct Fixup\_Block FixupBlock;

#define TARGETPROC "svchost.exe"

typedef struct \_PROCINFO  
{  
   DWORD baseAddr;  
   DWORD imageSize;  
} PROCINFO;

BOOL EXPD = FALSE;  
CHAR \*PID;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function reads the MZ, PE, PE extended and Section Headers from an EXE file.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// 解析PE文件，得到 PE 结构  
//  
BOOL readPEInfo(FILE \*fp, MZHeader \*outMZ,PE\_Header \*outPE,PE\_ExtHeader \*outpeXH,SectionHeader \*\*outSecHdr)  
{  
   MZHeader mzH;  
   long fileSize;  
   PE\_Header peH;  
   PE\_ExtHeader peXH;  
   SectionHeader \*secHdr;  
    
   fseek(fp, 0, SEEK\_END);  
   fileSize = ftell(fp);  
   fseek(fp, 0, SEEK\_SET);

   if(fileSize < sizeof(MZHeader))  
   {  
      printf("File size too small\n");       
      return FALSE;  
   }

   // read MZ Header  
   fread(&mzH, sizeof(MZHeader), 1, fp);

   if(mzH.signature != 0x5a4d)      // MZ  
   {  
      printf("File does not have MZ header\n");  
      return FALSE;  
   }

   printf("Offset to PE Header = %X\n", mzH.offsetToPE);

   if((unsigned long)fileSize < mzH.offsetToPE + sizeof(PE\_Header))  
   {  
      printf("File size too small\n");       
      return FALSE;  
   }

   // read PE Header  
   fseek(fp, mzH.offsetToPE, SEEK\_SET);  
   fread(&peH, sizeof(PE\_Header), 1, fp);

   printf("Size of option header = %d\n", peH.sizeOfOptionHeader);  
   printf("Number of sections = %d\n", peH.numSections);

   if(peH.sizeOfOptionHeader != sizeof(PE\_ExtHeader))  
   {  
      printf("Unexpected option header size.\n");  
       
      return FALSE;  
   }

   // read PE Ext Header  
   fread(&peXH, sizeof(PE\_ExtHeader), 1, fp);

   printf("Import table address = %X\n", peXH.importTableAddress);  
   printf("Import table size = %X\n", peXH.importTableSize);  
   printf("Import address table address = %X\n", peXH.importAddressTableAddress);  
   printf("Import address table size = %X\n", peXH.importAddressTableSize);

   // read the sections  
   secHdr = (SectionHeader\*)malloc( sizeof(SectionHeader)\* (peH.numSections) );

   fread(secHdr, sizeof(SectionHeader) \* peH.numSections, 1, fp);

   \*outMZ = mzH;  
   \*outPE = peH;  
   \*outpeXH = peXH;  
   \*outSecHdr = secHdr;

   return TRUE;  
}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function calculates the size required to load an EXE into memory with proper alignment.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// 返回文件所占用的内存空间  
//  
int calcTotalImageSize(MZHeader \*inMZ, PE\_Header \*inPE, PE\_ExtHeader \*inpeXH,SectionHeader \*inSecHdr)  
{  
   int result = 0;  
   int val, i;  
   int alignment = inpeXH->sectionAlignment;

   if(inpeXH->sizeOfHeaders % alignment == 0)   // PE头对齐  
      result += inpeXH->sizeOfHeaders;  
   else  
   {  
      val = inpeXH->sizeOfHeaders / alignment;  
      val++;  
      result += (val \* alignment);  
   }

   for(i = 0; i < inPE->numSections; i++) // 节对齐  
   {  
      if(inSecHdr[i].virtualSize)  
      {  
        if(inSecHdr[i].virtualSize % alignment == 0)  
           result += inSecHdr[i].virtualSize;  
        else  
        {  
           int val = inSecHdr[i].virtualSize / alignment;  
           val++;  
           result += (val \* alignment);  
        }  
      }  
   }

   return result;  
}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function calculates the aligned size of a section  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// 返回真实在内存中占用的大小  
//  
unsigned long getAlignedSize(unsigned long curSize, unsigned long alignment)  
{    
   if(curSize % alignment == 0)  
      return curSize;  
   else  
   {  
      int val = curSize / alignment;  
      val++;  
      return (val \* alignment);  
   }  
}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function loads a PE file into memory with proper alignment.  
// Enough memory must be allocated at ptrLoc.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// 加载PE文件到内存中  
//  
BOOL loadPE(FILE \*fp, MZHeader \*inMZ, PE\_Header \*inPE, PE\_ExtHeader \*inpeXH,SectionHeader \*inSecHdr, LPVOID ptrLoc)  
{  
   unsigned long headerSize, readSize;  
   int i;  
   char \*outPtr = (char \*)ptrLoc;

   fseek(fp, 0, SEEK\_SET);  
   headerSize = inpeXH->sizeOfHeaders;

   // certain PE files have sectionHeaderSize value > size of PE file itself.  
   // this loop handles this situation by find the section that is nearest to the  
   // PE header.  
//  
// 如果文件太小，以至与PE头中还包括了节的内容，这样就先不拷贝节的内容  
// 当然这种情况很少见  
//  
   for(i = 0; i < inPE->numSections; i++)  
   {  
      if(inSecHdr[i].pointerToRawData < headerSize)  
        headerSize = inSecHdr[i].pointerToRawData;  
   }

   // read the PE header  
   readSize = fread(outPtr, 1, headerSize, fp);  
   printf("HeaderSize = %d\n", headerSize);  
   if(readSize != headerSize)  
   {  
      printf("Error reading headers (%d %d)\n", readSize, headerSize);  
      return FALSE;       
   }

   //  
   // getAlignedSize 返回真实占用的内存的大小  
   //  
   outPtr += getAlignedSize(inpeXH->sizeOfHeaders, inpeXH->sectionAlignment);

   // read the sections  
   for(i = 0; i < inPE->numSections; i++)  
   {  
      if(inSecHdr[i].sizeOfRawData > 0)  
      {  
        unsigned long toRead = inSecHdr[i].sizeOfRawData;  
        if(toRead > inSecHdr[i].virtualSize)  
           toRead = inSecHdr[i].virtualSize;

        fseek(fp, inSecHdr[i].pointerToRawData, SEEK\_SET);  
        readSize = fread(outPtr, 1, toRead, fp);

        if(readSize != toRead)  
        {  
           printf("Error reading section %d\n", i);  
           return FALSE;  
        }  
        outPtr += getAlignedSize(inSecHdr[i].virtualSize, inpeXH->sectionAlignment);  
      }  
      else  
      {  
        // this handles the case where the PE file has an empty section. E.g. UPX0 section  
        // in UPXed files.

        if(inSecHdr[i].virtualSize)  
           outPtr += getAlignedSize(inSecHdr[i].virtualSize, inpeXH->sectionAlignment);  
      }  
   }

   return TRUE;  
}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function loads a PE file into memory with proper alignment.  
// Enough memory must be allocated at ptrLoc.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void doRelocation(MZHeader \*inMZ, PE\_Header \*inPE, PE\_ExtHeader \*inpeXH,  
            SectionHeader \*inSecHdr, LPVOID ptrLoc, DWORD newBase)  
{  
    long delta;  
    int numEntries,i, relocType;  
    unsigned short \*offsetPtr;  
    DWORD \*codeLoc;  
   FixupBlock \*fixBlk;

   if(inpeXH->relocationTableAddress && inpeXH->relocationTableSize)  
   {  
      fixBlk = (FixupBlock \*)((char \*)ptrLoc + inpeXH->relocationTableAddress);  
      delta = newBase - inpeXH->imageBase;

      while(fixBlk->blockSize)  
      {  
        printf("Addr = %X\n", fixBlk->pageRVA);  
        printf("Size = %X\n", fixBlk->blockSize);

        numEntries = (fixBlk->blockSize - sizeof(FixupBlock)) >> 1;  
        printf("Num Entries = %d\n", numEntries);

        offsetPtr = (unsigned short \*)(fixBlk + 1);

        for(i = 0; i < numEntries; i++)  
        {  
           codeLoc = (DWORD \*)((char \*)ptrLoc + fixBlk->pageRVA + (\*offsetPtr & 0x0FFF));  
            
           relocType = (\*offsetPtr & 0xF000) >> 12;  
            
           printf("Val = %X\n", \*offsetPtr);  
           printf("Type = %X\n", relocType);

           if(relocType == 3)  
              \*codeLoc = ((DWORD)\*codeLoc) + delta;  
           else  
           {  
              printf("Unknown relocation type = %d\n", relocType);  
           }  
           offsetPtr++;  
        }

        fixBlk = (FixupBlock \*)offsetPtr;  
      }  
   }    
}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// Creates the original EXE in suspended mode and returns its info in the PROCINFO structure.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

       
BOOL createChild(PPROCESS\_INFORMATION pi,        // OUT  
                PCONTEXT ctx,                    // OUT  
                PROCINFO \*outChildProcInfo        // OUT  
)  
{  
    PROCINFO \*outChildProcInfo2 = NULL;  
   STARTUPINFO si = {0};  
   DWORD read;  
   DWORD \*pebInfo;  
   DWORD curAddr;  
   MEMORY\_BASIC\_INFORMATION memInfo, memInfo2;  
   DEBUG\_EVENT DBEvent;  
   DWORD read2, curAddr2;  
   DWORD \*pebInfo2;

   if(!EXPD)  
   {  
        if(CreateProcess(NULL,  
            TARGETPROC,  
              NULL,  
            NULL,  
            0,  
            CREATE\_SUSPENDED,  
            NULL,  
            NULL,  
            &si,  
            pi))       
        {  
      ctx->ContextFlags=CONTEXT\_FULL;  
      GetThreadContext(pi->hThread, ctx);

// // 获取外壳进程运行状态，[ctx.Ebx+8]内存处存的是外壳进程的加载基址，ctx.Eax存放有外壳进程的入口地址  
      pebInfo = (DWORD \*)ctx->Ebx;  
      ReadProcessMemory(pi->hProcess, &pebInfo[2], (LPVOID)&(outChildProcInfo->baseAddr), sizeof(DWORD), &read);  
    
      curAddr = outChildProcInfo->baseAddr;

    //在 SVCHOST.EXE中寻找 MEM\_FREE 的内存地址  
      while(VirtualQueryEx(pi->hProcess, (LPVOID)curAddr, &memInfo, sizeof(memInfo)))  
      {  
        if(memInfo.State == MEM\_FREE)  
           break;  
        curAddr += memInfo.RegionSize;  
      }  
      outChildProcInfo->imageSize = (DWORD)curAddr - (DWORD)outChildProcInfo->baseAddr;

      return TRUE;  
   }  
   }  
   else{  
    if(DebugActiveProcess((DWORD)\*PID))  
    {  
       WaitForDebugEvent(&DBEvent,INFINITE);  
       pi->hThread=DBEvent.u.CreateProcessInfo.hThread;  
       pi->hProcess=DBEvent.u.CreateProcessInfo.hProcess;  
         ctx->ContextFlags=CONTEXT\_FULL;  
      GetThreadContext(pi->hThread, ctx);  
      pebInfo2 = (DWORD \*)ctx->Ebp;  
      \*pebInfo2+=0x30;  
      ReadProcessMemory(pi->hProcess, &pebInfo2[2], (LPVOID)&(outChildProcInfo2->baseAddr), sizeof(DWORD), &read2);  
    
      curAddr2 = outChildProcInfo2->baseAddr;  
      while(VirtualQueryEx(pi->hProcess, (LPVOID)curAddr2, &memInfo2, sizeof(memInfo2)))  
      {  
        if(memInfo2.State == MEM\_FREE)  
           break;  
        curAddr2+= memInfo2.RegionSize;  
      }  
      outChildProcInfo2->imageSize = (DWORD)curAddr2 - (DWORD)outChildProcInfo2->baseAddr;

      return TRUE;  
      }  
   }  
    
   return FALSE;  
}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// Returns TRUE if the PE file has a relocation table  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BOOL hasRelocationTable(PE\_ExtHeader \*inpeXH)  
{  
   if(inpeXH->relocationTableAddress && inpeXH->relocationTableSize)  
   {  
      return TRUE;  
   }  
   return FALSE;  
}

typedef DWORD (WINAPI \*PTRZwUnmapViewOfSection)(IN HANDLE ProcessHandle, IN PVOID BaseAddress);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// To replace the original EXE with another one we do the following.  
// 1) Create the original EXE process in suspended mode.  
// 2) Unmap the image of the original EXE.  
// 3) Allocate memory at the baseaddress of the new EXE.  
// 4) Load the new EXE image into the allocated memory.  
// 5) Windows will do the necessary imports and load the required DLLs for us when we resume the suspended  
//   thread.  
//  
// When the original EXE process is created in suspend mode, GetThreadContext returns these useful  
// register values.  
// EAX - process entry point  
// EBX - points to PEB  
//  
// So before resuming the suspended thread, we need to set EAX of the context to the entry point of the  
// new EXE.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void doFork(MZHeader \*inMZ,  
            PE\_Header \*inPE,  
            PE\_ExtHeader \*inpeXH,  
            SectionHeader \*inSecHdr, LPVOID ptrLoc,DWORD imageSize)  
{  
   STARTUPINFO si = {0};  
   PROCESS\_INFORMATION pi;  
   CONTEXT ctx;  
   PROCINFO childInfo;  
   LPVOID v;  
   DWORD oldProtect;  
   DWORD \*pebInfo;  
   DWORD wrote;  
   PE\_ExtHeader \*peXH;  
    
   if(createChild(&pi, &ctx, &childInfo))  
   {       
       pebInfo = (DWORD \*)ctx.Ebx;  
      printf("Original EXE loaded (PID = %d).\n", pi.dwProcessId);  
      printf("Original Base Addr = %X, Size = %X\n", childInfo.baseAddr, childInfo.imageSize);  
       
      v = (LPVOID)NULL;  
       
      if(inpeXH->imageBase == childInfo.baseAddr && imageSize <= childInfo.imageSize)  
      {  
        // if new EXE has same baseaddr and is its size is <= to the original EXE, just  
        // overwrite it in memory  
        v = (LPVOID)childInfo.baseAddr;  
        VirtualProtectEx(pi.hProcess, (LPVOID)childInfo.baseAddr, childInfo.imageSize, PAGE\_EXECUTE\_READWRITE, &oldProtect);         
         
        printf("Using Existing Mem for New EXE at %X\n", (unsigned long)v);  
      }  
      else  
      {  
        // get address of ZwUnmapViewOfSection  
        PTRZwUnmapViewOfSection pZwUnmapViewOfSection = (PTRZwUnmapViewOfSection)GetProcAddress(GetModuleHandle("ntdll.dll"), "ZwUnmapViewOfSection");

        // try to unmap the original EXE image  
        if(pZwUnmapViewOfSection(pi.hProcess, (LPVOID)childInfo.baseAddr) == 0)  
        {  
           // allocate memory for the new EXE image at the prefered imagebase.  
           v = VirtualAllocEx(pi.hProcess, (LPVOID)inpeXH->imageBase, imageSize, MEM\_RESERVE | MEM\_COMMIT, PAGE\_EXECUTE\_READWRITE);  
           if(v)  
              printf("Unmapped and Allocated Mem for New EXE at %X\n", (unsigned long)v);  
        }  
      }

      if(!v && hasRelocationTable(inpeXH))  
      {  
        // if unmap failed but EXE is relocatable, then we try to load the EXE at another  
        // location  
        v = VirtualAllocEx(pi.hProcess, (void \*)NULL, imageSize, MEM\_RESERVE | MEM\_COMMIT, PAGE\_EXECUTE\_READWRITE);  
        if(v)  
        {  
           printf("Allocated Mem for New EXE at %X. EXE will be relocated.\n", (unsigned long)v);

           // we&#39;ve got to do the relocation ourself if we load the image at another  
           // memory location            
           doRelocation(inMZ, inPE, inpeXH, inSecHdr, ptrLoc, (DWORD)v);  
        }  
      }

      printf("EIP = %X\n", ctx.Eip);  
      printf("EAX = %X\n", ctx.Eax);  
      printf("EBX = %X\n", ctx.Ebx);      // EBX points to PEB  
      printf("ECX = %X\n", ctx.Ecx);  
      printf("EDX = %X\n", ctx.Edx);  
       
      if(v)  
      {         
        printf("New EXE Image Size = %X\n", imageSize);  
         
        // patch the EXE base addr in PEB (PEB + 8 holds process base addr)  
             
        WriteProcessMemory(pi.hProcess, &pebInfo[2], &v, sizeof(DWORD), &wrote);

        // patch the base addr in the PE header of the EXE that we load ourselves  
        peXH = (PE\_ExtHeader \*)((DWORD)inMZ->offsetToPE + sizeof(PE\_Header) + (DWORD)ptrLoc);  
        peXH->imageBase = (DWORD)v;  
         
        if(WriteProcessMemory(pi.hProcess, v, ptrLoc, imageSize, NULL))  
        {    
           printf("New EXE image injected into process.\n");

           ctx.ContextFlags=CONTEXT\_FULL;            
           //ctx.Eip = (DWORD)v + ((DWORD)dllLoaderWritePtr - (DWORD)ptrLoc);  
            
           if((DWORD)v == childInfo.baseAddr)  
           {  
              ctx.Eax = (DWORD)inpeXH->imageBase + inpeXH->addressOfEntryPoint;      // eax holds new entry point  
           }  
           else  
           {  
              // in this case, the DLL was not loaded at the baseaddr, i.e. manual relocation was  
              // performed.  
              ctx.Eax = (DWORD)v + inpeXH->addressOfEntryPoint;      // eax holds new entry point  
           }

           printf("\*\*\*\*\*\*\*\*> EIP = %X\n", ctx.Eip);  
           printf("\*\*\*\*\*\*\*\*> EAX = %X\n", ctx.Eax);

           SetThreadContext(pi.hThread,&ctx);

           ResumeThread(pi.hThread);  
           printf("Process resumed (PID = %d).\n", pi.dwProcessId);  
        }  
        else  
        {  
           printf("WriteProcessMemory failed\n");  
           TerminateProcess(pi.hProcess, 0);  
        }  
      }  
      else  
      {  
        printf("Load failed. Consider making this EXE relocatable.\n");  
        TerminateProcess(pi.hProcess, 0);  
      }  
   }  
   else  
   {  
      printf("Cannot load %s\n", TARGETPROC);  
   }  
}

int main(int argc, char\* argv[])  
{  
    MZHeader mzH;  
    PE\_Header peH;  
    PE\_ExtHeader peXH;  
    SectionHeader \*secHdr;  
    LPVOID ptrLoc;  
    FILE \*fp;

   if((argc < 2 )||(argc > 3))  
   {  
      printf("\nUsage: %s [pid]\n", argv[0]);  
      return 1;  
   }  
      if(argc==3){  
        PID = malloc(1024);  
        memset(PID,0,1024);  
        strcpy(PID,argv[2]);  
        EXPD= TRUE ;  
      }  
       
   fp = fopen(argv[1], "rb");  
   if(fp)  
   {  
      if(readPEInfo(fp, &mzH, &peH, &peXH, &secHdr)) // 得到PE 结构  
      {  
        int imageSize = calcTotalImageSize(&mzH, &peH, &peXH, secHdr); //得到文件占用的内存空间的大小  
        printf("Image Size = %X\n", imageSize);

        ptrLoc = VirtualAlloc(NULL, imageSize, MEM\_COMMIT, PAGE\_EXECUTE\_READWRITE); //分配内存  
        if(ptrLoc)  
        {  
           printf("Memory allocated at %X\n", ptrLoc);  
           loadPE(fp, &mzH, &peH, &peXH, secHdr, ptrLoc);    //把文件加载到内存中                             
            
           doFork(&mzH, &peH, &peXH, secHdr, ptrLoc, imageSize);                      
        }  
        else  
           printf("Allocation failed\n");  
      }

      fclose(fp);  
   }  
   else  
      printf("\nCannot open the EXE file!\n");

   return 0;  
}